MIT LIBRARIES DUPL 2

3 9080 00579053 7

MASSACHUSETTS INSTITUTE of TECHNOLOGY
LIBRARIES

MASS. INST. TECH.
JAN 12 1990
LIBRARIES

CENTER FOR COMPUTATIONAL RESEARCH
IN ECONOMICS AND MANAGEMENT SCIENCE

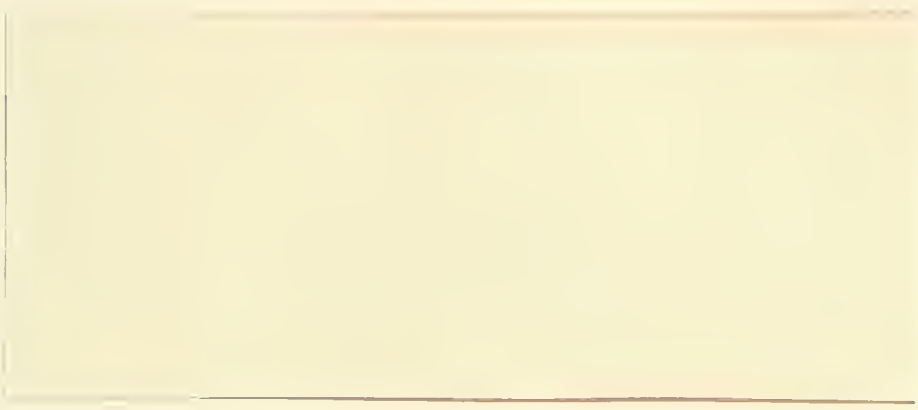
Factory Concepts and Practices in Software Development: An Historical Overview

Michael A. Cusumano

Mitsubishi Career Development
Assistant Professor of Management

Working Paper #3095-89 BPS
Version: December 5, 1989

ALFRED P. SLOAN SCHOOL OF MANAGEMENT
MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139



Factory Concepts and Practices in Software Development: An Historical Overview

Michael A. Cusumano

Mitsubishi Career Development
Assistant Professor of Management

Working Paper #3095-89 BPS
Version: December 5, 1989

Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139

(617) 253-2574

MIT LIBRARIES
JAN 12 1990
RECEIVED

ABSTRACT

This paper reviews the introduction of factory concepts and practices, based on tools and methods from the evolving field of software engineering, at major software producers, in particular those that explicitly adopted the factory label to describe their software facilities or approach to software development: Hitachi, Toshiba, NEC, and Fujitsu in Japan, as well as System Development Corporation in the U.S. The other U.S. firm discussed in detail is IBM, which, without adopting the factory label, introduced numerous measures to organize and control software development, especially basic software. The paper emphasizes that the difficulty of the technology, shortages of skilled engineers, and large-scale projects have encouraged producers to become more systematic or factory-like in managing a series of projects, even though some characteristics of the technology and the industry have made software seem difficult to control and more suitable to a loosely structured project-centered or craft approach to development.

INTRODUCTION

When analyzing the appearance of a new technology and the birth of a new industry, managers should find an historical perspective useful to refine strategies both for competitive positioning and technology management. For example, what and when might concepts or practices learned in one industry for managing a particular technology apply in other settings? Many industries have evolved beyond an initial phase, where product design and production remained loosely structured, expensive, and conducted by highly skilled individuals in a "craft" or job-shop mode, to more systematically organized modes of engineering and manufacturing. Product and process innovations have thus made a wide variety of sophisticated goods available to masses of consumers by systematically dividing large tasks into smaller and less-skilled operations, establishing standard procedures and controls to help de-skill tasks, apportioning labor among different individuals, building tools and other equipment to mechanize or automate aspects of design and production, and creating designs based on reusable or interchangeable components (Woodward 1965; Chandler 1977; Abernathy 1978; Hounshell 1984).

Some researchers, however, have argued that structured engineering or factory processes of this sort have become possible only in industries when the rate of product innovations or changes decline to the point where firms can concentrate on process innovations related to cost reductions (Abernathy and Utterback 1978, 1982). This requirement suggests that not all products or development processes are suitable for factory-oriented design and production systems, even though managers may have strong incentives to pursue more efficient modes of operations to compete effectively, especially in new industries with complex technologies.

The subject of this article is how major producers of software have

attempted to push forward the state of a new technology and thereby move software from loosely organized craft or job-shop modes of development to a more structured and automated process of engineering and production management. The number of software producers is too large to cover all firms adequately. This discussion, therefore, focuses on companies that, between the 1960s and 1980s, pioneered large-scale programming for commercial sale or explicitly introduced factory concepts and practices into their software-development operations: International Business Machines (IBM) and System Development Corporation (SDC) in the U.S., and Hitachi, Toshiba, NEC, and Fujitsu in Japan.

THE SOFTWARE PROBLEM

Writing software -- instructions required to operate programmable computers, first introduced commercially during the 1950s -- has plagued engineers, managers, and customers since the beginning of the computer era. The development process consists of requirements analysis, system design, detailed program design, coding, testing, and installation, as well as redesign or repairs referred to as maintenance. Yet these phases are usually more iterative than sequential, and often unpredictable in time and costs, because the productivity of individual programmers tends to vary enormously and depend on elements difficult for management to control, such as personal talent and experience with particular applications.

Software producers thus encounter budget and schedule overruns as the rule rather than the exception, especially when attempting to build large complex systems with many components for the first time. The sheer difficulty of software design and programming, exacerbated by a demand for programs

rising faster than the supply of software engineers, led to a situation referred to as the "software crisis" as long ago as 1969 (Hunke 1981; Frank 1983). Despite years of advances in tools (specialized programs and databases that aid in the production of other software), design and programming techniques, and products themselves, software has remained a most vexing challenge to all concerned, with many problems that plagued pioneers still persisting through the 1980s. In fact, continuing difficulties in software have led academic researchers and practitioners alike to argue that writing software is and may forever remain more like an art or a craft rather than evolve into a technology suited to the precise controls of "true" scientific, engineering, or manufacturing disciplines (Brooks 1975; Shooman 1983; Hauptman 1986; Mahoney 1988).

Indeed, software appears to have characteristics that make conventional engineering or factory operations difficult to introduce -- little product or process standardization to support economies of scale, wide variations in project contents and work flows, cumbersome tasks difficult and sometimes counterproductive to divide, de-skill, or automate. What is more, software producers have struggled to contend with constant evolution in product and process technology as well as in customer requirements, often for programs with unique or tailored features.

To manage such a complex technology in such a dynamic industry, many software producers have resorted to a simple solution: They have tried to remain flexible in structure and processes, hiring as many experienced or talented programmers as possible and relying on small, integrated teams -- as in job-shop or craft production in other industries. This approach takes advantage of the greater productivity usually associated with experienced personnel, and reduces coordination or communications problems within the project, as well as the need to divide labor and tasks too rigidly (Borum 1987).

Job-shop or craft practices have worked well in software and proved essential in the early days of the industry, when all products seemed new and changing, and when programs remained small, reflecting hardware limitations. But the software market continues to exhibit demand exceeding the supply of skilled programmers and managers, tight budgets and short schedules, customer requirements for unique features as well as low prices and high reliability, long-term maintenance costs surpassing those of new design work, and increases in the length and complexity of programs as hardware improves (U.S. Department of Commerce 1984; Schware 1989). These conditions have created huge incentives for managers to reduce skill requirements and systematically recycle key factors of production (methods, tools, procedures, engineers, components) in as many projects as possible (Ramamoorthy et al. 1984; Jones 1986; Freeman 1987). Thus, as the industry has evolved, loosely structured processes and craft organizations have become inadequate to manage software development, at least for many producers.

EARLY FRUSTRATIONS AND FACTORY CONCEPTS

The earliest public proposals for the introduction of factory-type methods, tools, and organizations to software development appeared in the late 1960s, as outgrowths of comparisons of programming with established engineering and manufacturing processes. An engineer at General Electric, R.W. Bemer, made numerous suggestions that culminated in a 1968 paper encouraging GE to develop a "software factory" to reduce variability in programmer productivity through standardized tools, a computer-based interface, and an historical database useful for financial and management controls. GE's exit from the computer business in 1970 ended the company's commitment to commercial

hardware and software production, although Bemer provided for the industry the first working definition of what might constitute a software factory:

> [A] software factory should be a programming environment residing upon and controlled by a computer. Program construction, checkout and usage should be done entirely within this environment, and by using the tools contained in the environment... A factory...has measures and controls for productivity and quality. Financial records are kept for costing and scheduling. Thus management is able to estimate from previous data... Among the tools to be available in the environment should be: compilers for machine-independent languages; simulators, instrumentation devices, and test cases as accumulated; documentation tools -- automatic flow-charters, text editors, indexers; accounting function devices; linkage and interface verifiers; code filters (and many others) (Bemer 1969: 1626-1627).

While Bemer focused on standardized tools and controls, Dr. M.D. McIlroy of AT&T emphasized another factory-like concept -- systematic reusability of code when constructing new programs. In an address at a 1968 NATO Science Conference on software engineering, McIlroy argued that the division of software programs into modules offered opportunities for "mass production" methods. He then used the term "factory" in the context of facilities dedicated to producing parameterized families of software parts or routines that would serve as building blocks for tailored programs reusable across different computers (McIlroy 1969). But reception to McIlroy's ideas was mixed: It seemed too difficult to create program modules that would be efficient and reliable for all types of systems and which did not constrain the user.

Software was also heavily dependent on the specific characteristics of hardware. Nor did anyone know how to catalog program modules so they could be easily found and reused (Horowitz and Munson 1984). Nonetheless, by the late 1960s, the term factory had arrived in software and was being associated with computer-aided tools, management control systems, as well as modularization and reusability.

The comments of Bemer and McIlroy reflected the growing frustrations of managers that had to oversee the production of large-scale programs. One of the most comprehensive discussions on software engineering, and still a most useful treatment, was a 1968 NATO Science Committee conference report. Managers and academics from several countries agreed that difficulties almost always began with the first step -- determining system requirements. Because customers often did not know exactly what type of software they would need, designers could not be certain what type of system to build. This uncertainty led to other difficulties as a project progressed: The initial cost and schedule estimates had to be changed, while programming modifications introduced during system development usually led to configuration, testing, documentation, and maintenance errors. In addition, the NATO committee thought it inherently problematic to manage large projects, given huge variations in programming productivity, communication problems within projects, lack of tools, lack of measurement standards, rapid growth in demand and size of programs, lack of standardization, few reusable components, and enormous maintenance costs as systems required repairs or changes over time (Table 1).

A comparison of the NATO report to numerous anecdotal accounts and formal surveys during the 1970s and 1980s confirms the persistance of these same problems (Boehm 1976; Thayer 1979; Arden 1980; Abdel-Hamid 1984; Ramamoorthy et al. 1984; Brooks 1987). But perhaps the most famous account

of problems in software development has been **The Mythical Man-Month**, published in 1975 by Frederick P. Brooks, Jr. In recounting his experiences managing production of the main operating system for IBM's System/360 family of mainframes in the mid-1960s, Brooks concluded, as did others before and after, that difficulties in software development were interrelated, nearly impossible to isolate, and thus inherently systemic. Most vexing to him were planning, project control, quality management, and maintenance. Brooks was particularly eloquent in arguing that poorly developed estimating techniques and methods to monitor schedule progress made projections based on "man-months" so inaccurate as to be practically "mythical" in content. He also claimed that the difficulty of achieving "efficiency and conceptual integrity" in a large project with many participants could be described by an "n-square law" effect, where n objects had $n(n-1)$ possible interactions. Brooks left readers with frightening imagery, comparing the task of large-scale software development to prehistoric animals struggling futilely to escape an unforgiving tar pit and sinking deeper with each movement:

> No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and sabertoothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.
>
> Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems -- few have met goals, schedules, and budgets. Large and small, massive or wiry,

team after team has become entangled in the tar. No one thing seems to cause the difficulty -- any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it (Brooks 1975: 4).

WHY SOFTWARE IS DIFFICULT

Why software development remained difficult to manage appeared to result from several interrelated factors, not all of which were technological in nature. Many problems seemed to stem from *the inherent complexity of designing, building, and testing computer code,* especially the unpredictability of component interactions as programs become large in size; the different ways of writing similar functions; and the near impossibility of fully checking all possible interactions of components and operating conditions. Software producers employed countermeasures, such as high-level languages, structured programming and design techniques, and a variety of support tools and documentation procedures. It also proved possible to reduce variability or the uncertainty of estimates of human performance by standardizing skills through training programs, providing everyone with the same tools and methods, keeping detailed data on the performance of specific individuals in different types of projects as well as data on the average performance of individuals with certain years of experience or training. Reuse of designs and code could further lessen both complexity and variability by cutting the amount of new software developers had to write.

These kinds of solutions -- systematic collection and utilization of individual performance data; standardized training, tools, and methods; and reusability -- appear frequently in engineering and manufacturing operations in other industries. But software managers still cannot predict what the process will be without some form of control or enforcement of standards and procedures that might not suit all projects and development personnel. In this sense, to what degree a particular firm or project might be able to structure its process for software development was not simply a technological problem. It depended on the kind and variability of projects as well as the appropriateness of tools and methods given market needs, technological changes, and the preferences of individual managers, engineers, and customers. These factors remained difficult to predict and control since they were both external and internal to the firm.

Diversity in user-requirements and applications presented another problem that put practical limits on reusability and repeatability of procedures, tools, or code from project to project, and raised the potential risks of standardizing designs, tools, methods, or training. Variability appeared necessary to accommodate different user needs and technological innovation. But when this reflected a lack of standards, poor planning and control, or insufficient product and market focus, managers probably could do more to limit the diversity they needed to manage, both in external market requirements and internal organizational practices.

At the same time, *the still-evolving state of both hardware and software technology*, especially the frequency of product and process innovations, made it difficult and risky for managers to focus and standardize operations too much. Especially in segments changing particularly rapidly, such as personal computers or engineering work stations, a prudent strategy might be to avoid investing in

standardization of methods, tools, or program libraries of reusable code that might soon become outdated.

Software developers themselves might also contribute to managerial and organizational constraints. They might insist on using their own methods and tools, despite company standards, or dislike filling out detailed reports needed for project control or process and quality analysis. In addition, they might prefer to build ever more sophisticated and innovative systems, rather than reusing specifications, designs, and code to make products similar to previous ones. In fact, one of the major difficulties cited in the 1968 NATO conference proceedings dealt with this last item, i.e., that software developers often preferred to write new programs, and thus they constantly immersed themselves (and the firm) in research and reduced their (and the firm's) ability to benefit from previous experience:

> Instead of trying to write a system which is just like last year's, only better implemented, one invariably tries to write an entirely new and more sophisticated system. Therefore you are in fact continually embarking on research, yet your salesmen disguise this to the customer as being just a production job.... This practice leads to slipped schedules, extensive rewriting, much lost effort, large numbers of bugs, and an inflexible and unwieldy product. It is unlikely that such a product can ever be brought to a satisfactory state of reliability or that it can be maintained and modified.... [T]his mixing of research, development, and production is a root case of the 'software gap'... (Naur and Randell 1969: 123).

Nevertheless, since the 1960s, large software producers have learned much about how and how not to manage software development, even though gains in programming productivity and quality have continued to trail advances in computer hardware by wide margins (U.S. Department of Commerce 1984). As companies attempted to become more systematic in software, they also moved closer to introducing factory-like concepts, tools, and organizational structures or processes. This occurred despite the tendency of software firms not to use the term factory to label their development facilities (except in Japan).

THE MARKET LEADER: IBM

For example, IBM, the largest producer of hardware and software in the world, preferring to use labels such as laboratory or programming center to described its many software facilities. Nonetheless, top management first began in 1960 to elevate basic software programming (though not applications development) to a level more comparable to hardware engineering, or at least to raise it above the status of a loosely controlled support activity. Programming until this time had been a relatively simple matter for IBM and customers in terms of personnel requirements and organization, since machines remained small and companies only needed a few people to write programs. But now discussions focused on the need to organize a tightly managed corporate-wide system for producing basic programs to run the new machines IBM planned to introduce and make sure new software did not contain the large number of quality problems reported in IBM's earlier programs.[1]

IBM did not fully centralize development operations but rather, over the next few years, set up a world-wide organization to produce the System/360 operating system (OS/360), probably the most difficult and expensive commercial

software project undertaken up to that time. The basic software included four different but "upward-compatible" versions of the operating system, each for machines of different sizes. OS/360 stood at the top of the line. Other less complex systems, providing basic programming capabilities through management of external disk and tape memory drives, were BPS (Basic Programming Support), BOS (Basic Operating System), DOS (Disk Operating System), and TOS (Tape Operating System).

IBM completed the smaller operating systems more or less on schedule, and they worked relatively well. OS/360, however, presented major difficulties. While IBM announced the System/360 family in 1964 and planned the first shipments for mid-1965, OS/360 was not ready, forcing IBM to ship the hardware with the simpler operating systems. After 5,000 man-years expended in design, construction, and documentation over a period of three years, including as many as 1000 employees working on the system simultaneously, IBM finally delivered a first version of OS/360 in 1966, containing huge numbers of defects that would require many more man-years to correct (Fisher et al. 1983).

The organization IBM used to produce System/360 software and later programs consisted of multiple development laboratories, each with hardware-engineering, programming, and marketing divisions or centers. Each laboratory took responsibility for different sizes of computers aimed at different customer markets, although each programming center had a formal mission or product-development assignment that contributed to a larger effort. A vice-president responsible for programming operations oversaw the different programming centers, and corporate IBM provided whatever financial resources managers needed for product development, tool and method R&D, and training. IBM distributed the development work among New York City (languages), Sweden (sort routines), Rochester, Minn. (processor control), San Jose (access methods),

Raleigh, N.C. (communications), Poughkeepsie, N.Y. (large operating system), and Endicott, N.Y. (mid-range operating system). Other centers in Kingston, N.Y., Germany, and additional locations also aided the effort.

This organizational structure reflected a strategy to create centers of experienced software developers around the world, specialized both in functions, such as sorts or communications software, and in machine types or sizes, such as mid-range versus large machines. Managers considered this dispersion necessary because of the logistical difficulty and expense of centralizing thousands of programmers. IBM, in the past, had already distributed hardware design, and management wanted to locate software personnel nearby hardware engineering for specific machines.

But dispersion and the independence of each programming site contributed to organizational as well as technical problems in completing System/360 software and subsequent products. For example, directors of each programming center, as well as project managers and individual engineers, retained considerable autonomy and the authority to decide what specific tools or methods they would use. Even as late as 1970, IBM had yet to establish a coordinated system for quality assurance and maintenance, and the lack of centralized control allowed the release of many important products with large numbers of defects, including OS/360 as well as a highly popular database management system. Simply correcting old problems consumed enormous expenses and manpower resources that directly detracted from new-product development. IBM also applied what managers later decided was the wrong organizational structure for software maintenance, a critical area. The company first patterned maintenance after hardware divisions, where mechanics individually went to customer sites and fixed problems as they occurred. In software, where thousands of customers might use a single package such as an

operating system or data-base management program, decentralized or individual maintenance proved to be far too labor intensive, costly, and unnecessary.[2]

But perhaps the major challenge IBM faced became the growing difficulty of coordinating large software projects dispersed around the world and then expecting groups to complete components on time that would work together according to technical specifications. IBM failed to do this not only with OS/360 but also with planned new versions of DOS/360, which IBM cancelled even before a public announcement. In addition, specifications for OS/370, announced along with the hardware in 1970, called for adding virtual memory capabilities to the basic operating system, and this required major technological changes in programming concepts that made new projects even more complex. Poor planning in another huge project to produce a successor to System/370, dubbed the FS for Future System, added to IBM's costs and man-power shortages. IBM management decided in 1974 to cancel this effort after spending $5 billion and using approximately half of all basic software personnel on the project, because FS would not run programs written for the 360 and 370 architectures and it seemed unlikely customers would accept the new system (DeLamarter 1986).

In an attempt to streamline and systematize its operations, IBM changed several practices, at least for basic-software development. One move, which some executives had demanded directly after the OS/360 experience, set up a system of strict individual accountability for new products, committing project managers to guarantee specific dates when their components would be ready. IBM also established an organization for what it called remote maintenance, with specialists in a centralized location fixing software before re-releasing it to customers. In addition, IBM began more centralization of development, though in the mid-1970s there remained as many as eight sites making

interdependent language compilers, database-management systems, and access programs. This situation persisted until IBM management decided to concentrate these programming activities in one location in California.[3]

In the meantime, IBM had been accumulating a wealth of knowledge regarding the management of software development that helped IBM and other firms introduce more systematic procedures and some support tools for testing and coding operations and then for design and process management. This evolution can be seen in an analysis of articles in the IBM Systems Journal, begun in 1962, which describe both actual practices in IBM as well as research. As the topical and chronological listings indicate (Tables 2 and 3), both research and practice in the 1960s and 1970s centered on testing, simulation and other forms of product evaluation. Articles on generic tools in the 1980s were more varied but now included design. In the methods area, most articles in the 1960s treated design, coding, and testing, with shifts away from coding in the 1970s and 1980s toward project management and programming environments, but still with considerable attention to design and testing .

With an extensive and continually growing basis of software-engineering knowhow, and rising recognition of the importance of software from its highest executives, IBM in the late 1970s took a further step in organizational and technological integration by creating a new software facility combining the latest tools, techniques, and concepts regarding programming environments: the Santa Teresa Laboratory. This facility, opened in 1977, brought together 2000 software personnel organized along functional product lines (languages, database systems, access methods) and provided a new physical setting, designed by the head of Harvard University's School of Design, Gerald McCue (McCue 1978). As its name implies, IBM executives did not view or label the Santa Teresa Laboratory as a factory. In practice, however, they created a highly

centralized, functionally organized facility, with a standardized process and physical layout designed to facilitate programmer performance, comfort, and communication. This consolidation, moreover, saved on overhead costs by eliminating duplication in many staff positions and gave management more immediate control over product development and testing, of new and existing products.

IBM in the 1980s operated other sites with several hundred and even over a thousand software personnel in one location, although management did not appear to duplicate the degree of consolidation at Santa Teresa (except for an applications facility in Tokyo with several thousand personnel on one site). On the one hand, moving over a thousand developers and their families from eight different cities to Santa Teresa became extremely expensive. Top executives decided not to invest in a similar effort unless it seemed absolutely necessary (as in Japan, where IBM's competitors also concentrated large numbers of systems engineers and programmers in integrated facilities). In addition, because of the sheer size of IBM's hardware development and programming operations, it seemed unlikely management would ever try to centralize development operations completely and risk losing the ability to tap expertise around the world, even though dispersed operations made it difficult to coordinate and standardize methods, tools, and controls.[4]

Equally important, IBM encountered practical limits to large-scale integration and consolidation of personnel in functional departments. For example, Santa Teresa first attempted a relatively strict functional organization, with large departments handling all coding or all design. Ultimately, however, IBM managers concluded this structure did not help them serve individual customers effectively when products required close coordination between design and product construction. Management thus shifted Santa Teresa toward a

mixture of functional and product-centered departments in the late 1970s and 1980s, with specialized design, coding, and testing groups within product-oriented divisions[5] -- a structure similar to Japanese software factories described below.

In addition to finding a better compromise between process or organizational efficiency and flexibility, IBM in the late 1980s also worked toward solving another issue: increasing its ability to standardize hardware and software internal architectures and interfaces. An example of this effort, IBM's Systems Application Architecture (SAA) initiative, when completed, would provide a uniform interface to make it easier to reuse portions of systems and tools as well as whole programs across previously incompatible machines. At one time, incompatible lines supported an effective marketing strategy aimed at meeting different market segments with different products and dissuading competitors from introducing an equally broad range of computers. IBM in the past also appeared to view applications software as an extension of sales and did not subject this area to the same discipline and structuring as basic software. But, by the mid-1980s, intensified competition from firms such as Digital Equipment Corporation, which offered compatible small and large machines that could use the same software and databases, had forced IBM to reevaluate its strategy and attempt tighter controls and integration.

THE FIRST U.S. SOFTWARE FACTORY: SDC

One of the U.S. leaders in the custom software field, System Development Corporation, formerly a part of the Rand Corporation and in 1988 a Unisys division, established the first U.S. software facility called a factory in 1975-1976. SDC had been separated from Rand in the 1950s to develop the SAGE

missile control system for the U.S. Department of Defense. It later took on other real-time programming tasks as a special government-sponsored corporation, but went public in 1970. Top management then had to control software costs and launched a process-oriented R&D effort in 1972 to tackle five problems SDC programmers continued to encounter project after project:

(1) Lack of discipline and repeatability or standardized approaches to the development process.

(2) Lack of an effective way to visualize and control the production process, as well as to measure before a project was completed how well code implemented a design.

(3) Difficulty in accurately specifying performance requirements before detailed design and coding, and recurrence of disagreements on the meaning of certain requirements, or changes demanded by the customer.

(4) Lack of standardized design, management, and verification tools, making it necessary to reinvent these from project to project.

(5) Little capability to reuse components, despite the fact that many application areas used similar logic and managers believed that extensive use of off-the-shelf software modules would significantly shorten the time required for software development (Bratman and Court 1975 and 1977).

Ultimately, the SDC engineers constructed a detailed factory plan that consisted of three elements: an integrated set of tools (program library,

project databases, on-line interfaces between tools and databases, and automated support systems for verification, documentation, etc.); standardized procedures and management policies for program design and implementation; and a matrix organization, separating high-level system design (at customer sites) for program development (at the Software Factory). The first site to utilize the factory system, which SDC copyrighted under the name "The Software Factory," was a facility of about 200 programmers in Santa Monica, California. Determining the final form of the factory, however, required years of effort as well as trial and error.

For example, SDC management and R&D engineers did not conceive of the factory initially as being more than a set of tools that they hoped SDC facilities around the country might use; in other words, they did not think of the Software Factory as a physical, centralized facility, similar to factories in other industries. One reason was that the Department of Defense and other customers frequently required software contractors to locate development teams at the computer sites. Other factors also had made a large, centralized facility impractical, although circumstances were changing.

The incompatibility of many computers for which SDC wrote programs, and the wide variety of applications in the jobs SDC accepted to keep its employees active, had made it logical for SDC and most other software producers to organize work in projects, with integrated teams building their own tools and deciding on practices suitable for each job. This craft or job-shop approach made it difficult to standardize practices among projects or eliminate redundancies in areas such as tool development, design, and coding, but it brought customers, systems engineers, and software developers together, and seemed to increase the likelihood that the software system would meet customer requirements. However, a growing shortage of skilled programmers made it

harder for managers to find the proper set of experts on components such as operating systems, compilers, interfaces, and telecommunications in different locations or to locate engineers willing to move frequently. By 1976, this shortage of people had prompted SDC management to consider an alternative organization.

After working with the R&D team for a year on tools and methodologies, the new leader of the project, Jack Munson, recommended that SDC create a single facility -- an actual factory -- to build software for SDC's System Division. This facility would locate specialists in one site while management would bring programming jobs to them. Engineers in the facility would use the tools and methods developed in R&D, as well as remote-terminal and computer technologies that allowed "a single set of personnel to monitor and control a large number of software projects concurrently, taking advantage of economies of scale and providing for cross-utilization of scarce skills" (Baum 1981: 220-223).

As in the past, the factory structure required program offices at each customer site. Program managers maintained responsibility throughout the life-cycle for project management, customer relations, requirements and performance specifications, systems engineering, and quality control and assurance. To build the actual software and test it, however, program managers had to transfer system specifications to what was essentially an assembly line of three groups within the factory: Computer Program Design, Computer Program Development, and System Test and Verification. The SDC team, at least in 1977, recognized that separating system design from program production offered both an advantage in maintain closeness between systems engineering and customers as well as a potential disadvantage in removing the program developers and testers from the customers and systems engineers. But they hoped that informal

communication and methodological standards -- "a normal, well-understood interface procedure" -- could overcome any problems (Bratman and Court 1977: 128).

SDC gave the name Factory Support System to the "basic structural and control components" designed to facilitate the factory methodology (Bratman and Court 1975: 30-36; Bratman and Court 1977: 128-137). This tool set, written in a high-level language to ease portability, ran on an IBM 370 mainframe computer and used the facilities of IBM's operating system to automate procedures for keeping track of program development and collecting data (Figure 1). The tools supported top-down development, automated several management activities, maintained requirements through implementation, provided a library history for programs, allowed for symbolic system-data control, and automated aspects of program inspection and qualification.

It also took a year and a half during 1975-1976 for the R&D team to identify standards and procedures -- general rules and specific guidelines-- that might be applied to a variety of software projects. They based the process around a life-cycle model of software development covering the major activities, events, and product components common to all projects. The methodology, codified in what SDC called the Software Development Manual or SDM, called for structured design and coding, top-down program development, and program production libraries. In addition, SDM outlined a management and control process, providing guidelines for planning, project control, review and evaluation procedures, and quality assurance (Baum 1981: 22).

The R&D team in part borrowed from existing U.S. military standards (Bratman and Court 1977: 120), but established most of the SDM methodology by examining previous projects SDC had done through written records and interviewing personnel to determine what had worked well and appeared to

represent "best practice" within the company. According to the key factory architects, Harvey Bratman and Terry Court, this effort was critical to creating a common language and methodology that would make the factory more than just a building with a large number of programmers and projects working from a common pile of tools:

> The procedures hold the disparate portions of the Factory together: the standards are the means of making the Factory efficient and easy to use and learn. Without the standards that establish the Software Factory methodology and the procedures that establish the precise way of doing a task, the Factory is little more than an agglomeration of software development tools. Standardization, initially established and faithfully maintained to reflect changing conditions and continual improvement, is essential to the success of the Factory. The ... standards...establish a working environment where the creative design solutions of key technical personnel can be implemented in high-quality products, on schedule, and within budget. More specifically they:
>
> -- promote repeatability in the software development process;
> -- allow rapid transfer of expertise from one project to another;
> -- establish a consistent framework for cost estimating;
> -- make it possible for people to interact efficiently and with a common understanding of goals;
> -- provide the capability to measure project progress in a realistic manner;
> -- enforce technical and management techniques;
> -- establish a basis for assuring and measuring software quality

(Bratman and Court 1977: 121).

Approximately 10 projects went through the SDC Software Factory between 1976 and 1978. Four of the largest were TIROS-N, a $7 million data-processing system completed in 1978 for the TIROS-N weather satellite; the Mobile Sea Range Scoring system for navy battle training; the Morocco Air Defense System (MADS), a $12 million contract for systems engineering, software development, and training, with Westinghouse as the prime contractor, based on SAGE and a successor design and described by SDC's company history as "the first airspace system to use a standard commercial computer (the Burroughs B7700) and to be implemented in a standard higher-order language (ALGOL)...."; and the Emergency Command Control Communication System (ECCS), a $28.5-million project for the Los Angeles Police Department. The police system, referred to in the company history as "the largest and most complex police command-control system ever conceived," handled 3 million calls per year and one call every 4 seconds at peak loads, as well as 1.2 million vehicle dispatches per year, linking 850 mobile digital terminals in patrol cars as well as 5000 portable two-way communication devices (Baum 1981).

SDC managers claim that all the projects, with the exception of the system for the L.A. Police Department, came in on time and within budget, and with fewer defects and problems than SDC usually experienced with large systems. The company history also described the factory projects as, for the most part, "accurate, timely, and on budget," and models of "optimum software development" (Baum 1981: 205, 224). In fact, the factory worked so well that SDC's chief executive, George Mueller, directed all divisions to adopt the methodology as a corporate standard and in 1978 promoted the head of the factory project, Jack Munson, to oversee this transfer.[6]

Yet Munson's personal advancement contributed to the end of the factory, which fell gradually into disuse as the number of new projects going through the facility declined. The general lack of tool portability provided one reason why SDC's top executives began to lose interest in the factory concept. But problems with a particular project led Mueller to end his support and provided ammunition for program (project) managers who did not want to use the factory structure to build their software.[7] This was the ECCS system for the L.A. police.

According to Court, who headed the software-development effort for this project, ECCS broke down mainly from SDC's lack of experience in systems engineering for this type of application. In terms of writing and testing code, the job of personnel in the software factory, ECCS resembled other systems SDC had built. It was a real-time command and control application, with strict time and space constraints, and numerous communications and data-management requirements. However, the functionality of the application differed from anything SDC had done before: a police-department information system. Thus, in Court's words, "our systems engineering could not dominate the process, because they knew very little about police work and police operations. So the police tended to dominate the process, and as a result, the requirements kept on expanding and expanding." This led to an "endless discussion and analysis of requirements" that consumed nearly two years, throwing off the entire schedule and budget. Not surprisingly, the specifications SDC eventually agreed to turned out to be more complex and took longer to code than managers had anticipated, particularly since they tended to exceed the capabilities of the hardware (4 Digital PDP 11/70 minicomputers), making it necessary to refine the software to run as efficiently as possible. SDC thus lost a large sum of money on this project since it had agreed initially to a fixed-price contract.

To complete ECCS, SDC reverted to its former job-shop structure and created a dedicated project team of the best people available. This group finished the specifications as well as built and tested the software. Future jobs also went back to the project system, rather than using the factory structure for program development.[8] The factory methodology, embodied in the SDM manual, remained, although SDC dispersed the factory personnel among different projects and sites. SDC also updated the manual every few years and continued to use it through the late 1980s, while concepts from the factory methodology also spread to other firms after the U.S. Department of Defense contracted with SDC in 1976 to devise guidelines for military-use software procurement.[9]

Dissolution of the factory had both negative and positive consequences for SDC. On the one hand, dispersing programmers to different sites helped personnel work closely with customers and specialize in particular applications. Some former factory engineers also took on staff roles, helping different projects with proposals, new-business development, customer relations, strategic planning, and other tasks.[10] Dissolution of the factory tool set seemed a natural outcome to some managers, because of the dynamic nature of tool development in software and the incomplete state of what the factory had to offer. SDC had never completed all the tools and expected them to evolve, with old tools falling into disuse.

On the other hand, ending the factory did not help SDC solve the five problems that had inspired the factory's inception. The methodology continued to exist as a corporate standard, although it became more difficult to enforce good practices and use common tools across numerous independent sites. SDC also never developed the ability to reuse software systematically across different projects, and gradually declined from a position in the 1960s and early 1970s as a leader in the software industry, eventually merging with Burroughs

in 1981.

In addition, ten years after the end of the factory, at least some SDC divisions were still trying to build a common tool set and programming environment. In the 1980s, SDC management also appeared to be moving some facilities back toward centralization, but without separating systems engineering from program construction so rigidly and letting facilities focus on specialized lines of business. This seemed a better way to gain some scale economies despite the lack of more rationalized structure, like the original factory had attempted.

In retrospect, SDC may have attempted to impose a factory infrastructure of standardized tools and methods, and reusability goals, on a range of projects that were too different and better suited to job-shop production. The state of software technology made it difficult to transport tools and code across different machines. Furthermore, architects of the factory failed to solve the matrix-management problems that surfaced and prevented a steady work flow into the facility. Nonetheless, even though SDC abandoned the effort, its factory model influenced similar initiatives already underway in Japan at Hitachi as well as Japan's other leading computer producers (Table 4). While each differed in some respects, reflecting variations in company histories and the market focus of their software products, the Japanese efforts proved to be remarkably similar in technical emphases, organization, and management, as well as effective in improving levels of project control, productivity, quality, and reusability.

JAPANESE FACTORY EFFORTS

Hitachi

Hitachi, by founding its Software Works in 1969, was the first company in the world to apply the term factory (actually, its Japanese equivalent, *kojo*, translated either as "factory" or "works") to an actual software facility. A history of independent factories for each major product area prompted executives in the computer division to create a separate facility for software when this became a major activity. The objectives Hitachi managers set for their factory were to transform software from an unstructured "service" to a "product" with a guaranteed level of cost and quality, and use a centralized approach to achieve productivity and reliability improvements through process standardization and control.

A particular set of circumstances led Hitachi to establish a software factory. First, management saw a need to offset a severe shortage of skilled programmers in Japan and deal with numerous complaints from customers regarding defects in the software Hitachi was providing (most of which, along with the hardware, Hitachi was importing from RCA until 1970). Second, the fact that all Hitachi factories had to adopt corporate accounting and administrative standards forced software managers to analyze the software development process in great detail and experiment with a series of work standards and controls. The independence of Hitachi factories within the corporation also gave factory managers considerable authority over technology development.

Managers concentrated initially on determining factory standards for productivity and costs in all phases of development, based on standardized data bases for project management and quality control. Hitachi then standardized design around structured programming techniques in the early 1970s, and

reinforced these standards with training programs for new employees and managers. This reflected an attempt to standardize and improve average skills, rather than specify every procedure to be performed in each project and phase of development. The results: Late projects dropped from 72% in the first full year of the Software Works' operations in 1970 to approximately 7% in 1974, and remained under 19% in subsequent years, while the number of defects per machine in the field fell from an index of 100 in 1978 to approximately 13 during 1983-1985.[11]

Hitachi managers also underestimated how difficult implementing factory concepts such as reusability and process standardization would be. Two examples illustrate this. First, their attempt in the early 1970s to introduce a "components control system" for reusability failed. As one of the managers responsible for building the factory systems, Kanji Shibata, recalled: "Around 1970 we believed we had to introduce a components control system similar to what you find for hardware, and in the Software Works we established a committee to take this up as a special project." But the committee members soon realized that "software is not sufficiently standardized to be treated the same as in hardware components control." They changed their priorities and decided to find a way to standardize product designs, and then worry about components.

A survey of the programming field suggested that structured design and programming techniques would help standardize software design and coding. The committee renamed itself the Structured Programming Methods Committee and spent several years studying these techniques (as they were evolving) and analyzing programs Hitachi had already written. This was truly a pioneering move, because it would be several years before articles in industry journals began discussing structured design and programming widely and companies such

as IBM adopted these practices for their internal standards.[12]

Hitachi also attempted and failed to introduce one standardized process for both basic software and custom applications software. The need to separate methods and tools for different software types led to a separate division for basic software and applications within the Hitachi Software Works, and then to establishment of a second software factory, dedicated to applications only, in 1985. At the start of the factory, however, a committee for work standards took on the task of establishing procedures for all activities, based on a life-cycle model of development. This committee met almost weekly, studying available materials on software development and examining practices within Hitachi, and completed a first-cut set of standards by the end of 1970, referred to as the Hitachi Software Standards (HSS). These covered product planning, design and coding methodology, documentation and manuals, testing, and any other activities necessary to complete a software product. Although Hitachi managers clearly recognized these procedures and standards would have to evolve as personnel and technology changed, and they made provisions to revise some standards annually, drawing up the standards helped managers and engineers identify best practices within the company and within the industry for dissemination to all projects and departments.

Struggling with work standards also helped the committee recognize the need to distinguish between basic systems and applications software, rather than continue trying to impose similar controls and expectations on all types of software development. It started developing separate standards for applications during 1971-1972 and published system-consultation standard times in 1973. By 1975, the Software Works had completed an initial set of procedures now termed HIPACE (Hitachi Phased Approach for High Productive Computer System Engineering), to standardize formats for proposals, designs, and program

construction, as well as to aid in developing tools for design automation. The evolution of different standards and tools then made it relatively easy to establish a separate applications software factory in the 1980s.[13]

Another important juncture came in 1973, when management at the Software Works began training programmers in structured design and requiring new projects to follow structured concepts, to facilitate design standardization as well as maintenance and portability. Somewhat longer programs appear to have resulted, but managers considered the benefits to be significant.[14] At this point, the factory was also able to institute the components-control system that Hitachi had wanted several years earlier, to standardize procedures for module construction and configuration. Furthermore, this system helped simplify design and maintenance, and increased the potential portability of code, though reusability was not yet a major goal in Hitachi (Hitachi 1979: 117-118).

By the late 1970s, Hitachi had succeeded in organizing its software factory around a combination of manual engineering and factory techniques--structured design and coding coordinated with data collection and standard times for each activity, detailed documentation, design reviews independent of project personnel, rigorous defect analysis, and other elements (Table 5). At this point, after spending years studying and standardizing the development process, Hitachi management was ready to invest heavily in computer-aided tools, relying on engineers mainly from the Software Works and its new Systems Development Laboratory, part of the central laboratories structure.

The tools Hitachi adopted for its software factories were comprehensive, covering each function and activity in software development. For basic software, during the late 1970s, Hitachi introduced CASD (Computer-Aided Software Development System) to facilitate design, coding, documentation, and testing, and CAPS (Computer-Aided Production Control System for Software) to

manage manpower estimation, process flow control, and quality control. Both CASD and CAPS were continually evolving, as seen in Hitachi's gradual adoption in the late 1980s of a new system for project control, PDOC (Project Diagnostic Check System).

For custom applications, in the late 1970s, Hitachi began developing a set of tools and methodologies under the label ICAS (Integrated Computer-Aided Software Engineering System) (Kobayashi et al. 1983; Kobayashi and Aoyama 1986). The most important ones consisted of the SEWB (Software-Engineering Workbench), which supported both system design and programming on advanced work stations, and EAGLE (Effective Approach to Achieving High Level Software Productivity), which ran on Hitachi mainframes and helped programmers build software from reusable modules as well as structure new designs and code for reuse (Hagi et al. 1986). HIPACE continued to serve as the basic methodology used with these and other tools (Miyazoe et al. 1980).

As of 1989 (see Table 4), Hitachi operated the two largest software factories in Japan. The original Hitachi Software Works, with approximately 5000 personnel (including approximately 3000 assigned from Hitachi subsidiaries and subcontractors), built a range of basic software for Hitachi mainframes, including operating systems, language compilers, and database systems. The System Design Works (formerly called the Omori Software Works), housed 6000 software developers (including 4000 personnel from subsidiaries and subcontractors) in two 31-story towers. This facility, unlike some other Japanese applications software factories, combined systems engineers (those who designed the systems) with programmers that built the actual software, and concentrated on customized business applications systems such as for banking, inventory, or personnel control.

Toshiba

Toshiba created what is perhaps the most structured factory, although it also had the most focused product lines, using a centralized software facility to develop real-time process control software for industrial applications. The decision to establish a software factory stemmed from rapid increases in actual and projected demand, beginning around 1975, for industrial control systems relying on a new generation of relatively inexpensive minicomputers. Typical of the changing demands Toshiba faced as sales of its control minicomputers rose were orders from Japanese power-utility companies to develop automated thermal-power generating stations. These used enormous and growing amounts of software; the typical power-generation control system rose from a few hundred thousand lines of code in the mid-1970s to two million by the early 1980s, necessitating years of development and hundreds of new programmers.[15] Furthermore, to achieve safe and untended operation, the hardware and software for these and many other control systems had to be nearly free of defects or at least highly tolerant of system faults.

An R&D group responsible for software development in Toshiba, led by Dr. Yoshihiro Matsumoto, introduced an organization and process in 1977 integrating tools, methods, management and personnel systems, as well as a physical layout for work stations (Table 6). The strategy for utilizing this infrastructure centered around four policies: standardize the development process, to reduce variations among individuals and individual projects; reuse existing designs and code to build new systems, to reduce redundant work and maximize productivity; introduction of standardized and integrated tools to raise the performance levels of the average programmer; and provide extensive training and career-development tracks for programmers, to relieve the shortage of skilled engineers.

Perhaps the most delicate feature of the Toshiba Software Factory was its organizational structure, a matrix imposed over product departments from several operating groups and divisions, all located in the Fuchu Works. Toshiba referred to vertically-linked staff activities as its "product engineering system," horizontally-linked industrial-sector activities as its "production engineering system," and the overall arrangement as "engineering matrix management" (Shimoda 1986: 103-104).

Established in 1940 and set on 198 acres in the western outskirts of Tokyo, the Fuchu Works in 1987 had 7500 employees working primarily in four areas: Information Processing and Control Systems, Energy Systems, Industrial Equipment, and Semiconductors (Printed Circuit Board Division) (Toshiba 1987: 5). Operating departments within the divisions corresponded roughly to 19 product lines (Table 7). Each department contained sections for hardware and software design as well as for manufacturing (in the case of software, detailed design, coding, and module test), testing, quality assurance, and product control. Approximately half the Fuchu Works' employees were software developers, with a third working in the Software Factory at any given time, assigned from product departments for particular projects.[16] The recognition that some application areas required specialized systems analysis or engineering and design skills, and that all departments could benefit from sharing scarce software development resources, led to the adoption of this organization.

Similarities in the type of software the Fuchu Works built from project to project allowed Toshiba to deliver "semi-customized" programs that combined reusable designs and code with newly written modules, rather than writing all software from scratch. Toshiba also relied heavily on a standardized tool and methodology set, the Software Engineering Workbench (SWB), developed gradually after 1976. This utilized a version of the Unix operating system as

well as a full complement of tools for design-support, reusable module identification, code generation, documentation and maintenance, testing, and project-management (Table 8). Important features of the Toshiba methodology were the design of new program modules (generally limited to 50 lines) for reusability, the requirement that programmers deposit a certain number of reusable modules in a library each month, and the factoring in of reuse objectives into project schedules and budgets.

Table 9 shows gross software productivity for the Fuchu Works and the Toshiba Software Factory from 1972 to 1985, and reuse rates in the programming or coding phase as well as new code estimates from 1977 (the year the factory opened). Particularly striking is the rise in productivity (delivered equivalent assembler source lines or EASL per person per month, excluding operating systems, utilities, and other basic software) and the obvious impact of increasing reuse (measured at the code level). Productivity rose from 1390 lines per person per month in 1976 to over 3100 in 1985 (the last year for which Toshiba had made data public), while reuse levels (lines of delivered code taken from existing software) increased from 13% in 1979 to 48% in 1985. The 3130 lines of EASL source code per month per employee translate into approximately 1000 lines of Fortran, the most common language Toshiba used in 1985[17]--considerably more than the 200 to 300 lines of new code per month cited in various sources for U.S. programmers making similar real-time applications.[18] Quality levels (defined as the number of major faults detected after final testing) also improved dramatically after the opening of the factory, ranging from 7 to 20 per 1000 lines of delivered code converted to EASL (the range depended on the amount of testing customers contracted for) to .2 to .05 in 1985.

Toshiba data indicated that reusability was the major reason for

productivity and quality improvements. Management pursued reuse initially to accommodate rising demand and insure a high level of productivity and quality, despite variations in individual skills and diverse customer requirements, by building products from standardized components reassembled or combined with new components (Matsumura et al. 1987) This approach -- systematic creation and recycling of reusable software parts, as opposed to development from scratch or "accidental" reuse -- lay at the heart of Toshiba's concept of *factory* production of software. But how the factory carried out reuse in the face of long-standing organizational as well as technological problems provides a superb example of integrated management: of product planning, development techniques, computer-aided tools, and controls and incentives.

On the organizational side, writing and documenting components for reuse generally required extra time that individual customers might not want or should not want to pay for. Project managers, and project members, also have good reason to resent this extra time, unless they see opportunities to save time in the future. On the technical side, many factors influence whether a particular project can recycle existing components. These factors include the match in application or function of the existing software compared to the new design requirements; the particular language(s) used and the characteristics of the target computers; and the program structure and degree of modularity--how independent modules are of other modules or subsystems in the original program, based on the construction of the interfaces between various layers in a complete system (the new application software, subsidiary applications packages, utilities, the operating system, and the hardware).

Toshiba did not solve all problems related to reusability. Rather, the factory presented a system that simplified or restricted problems to manageable areas, and that provided incentives both for project managers and personnel to

write reusable software and reuse it frequently. Table 9 tracks the success of these efforts, indicating that approximately half the software Toshiba's factory delivered in 1985 consisted of reused code, including some with modifications (usually no more than 20% of the contents of an individual module). The other half of delivered software consisted of new code, mainly written for an individual plant or application, as well as some applications packages product departments developed to serve as major components of customized systems.

The organization Toshiba created to promote reuse and overcome short-term concerns of project managers and development personnel relied on Software Reusing Parts Steering Committees and a Software Reusing Parts Manufacturing Department and Software Reusing Parts Center (Figure 2). The factory formed a steering committee for different areas (with different members, depending on the application) to determine if customers had a common set of needs suitable for a package, and then allocated funds from the Fuchu Works' budget for these special projects. Some packages were utilities usable in different departments, although most served specific applications.[19]

The Reusing Parts Manufacturing Department and Parts Center evaluated new software (and documentation) to make certain it met factory standards; after certification, engineers registered the software in department or factory reuse databases (libraries). Registered items required a keyword phrase to represent the functionality of the part or correspond to a specific object, as well as a brief "Description for Reusers" (DFR) that explained the part's basic characteristics.

Management also relied on an integrated set of incentives and controls to encourage project managers and personnel to take the time to write reusable software parts and reuse them frequently. At the start of each project, managers agreed to productivity targets that they could not meet without

reusing a certain percentage of specifications, designs, or code. Design review meetings held at the end of each phase in the development cycle then checked how well projects met reuse targets, in addition to schedules and customer requirements. At the programmer level, when building new software, management *required* project members to register a certain number of components in the reuse databases, for other projects. Personnel also received awards for registering particularly valuable or frequently reused modules, and they received formal evaluations from superiors on whether they met their reuse targets. The SWB system, meanwhile, monitored reuse levels as well as deviations from targets both at the project and individual levels, and sent regular reports to managers (Matsumoto 1981; Matsumoto 1984; Matsumoto 1987).

NEC

The first step toward a factory structure for software development at NEC consisting of founding the Basic Software Development Division at its Fuchu Works in 1974, thereby separating organizationally operating-systems development from hardware development. NEC subsequently established other organizations at Mita, Tamagawa, and Abiko, all within the Tokyo metropolis or suburbs, for its other software needs, as well as dispersed programming work throughout Japan through numerous subsidiaries and satellite offices.[20] However, the separation and separate histories of these facilities gradually presented greater problems for managers pursuing standardization and common goals such as quality improvement throughout the NEC group. Table 10 and the discussion below summarizes the key initiatives NEC managers adopted to create a more effective multi-product, multi-process factory network.

The Software Strategy Project, started in 1976, attempted to integrate programming operations on a group-wide basis (including all in-house divisions

and subsidiaries, rather than just the computer division). The objective was to introduce standards for tools, procedures, and methodologies for all phases of development and all aspects of management. Yet it took several years of trial and error to accomplish this while allowing individuals, projects, and divisions sufficient flexibility to tailor standards to the needs of their products and customers.

When the Software Strategy Project ended, managers who worked on the effort, led by Dr. Yukio Mizuno, noticed another weakness in NEC's structure for managing software development: the lack of permanent staff to explore and follow through on key issues or technologies. Thus, to insure continuity and proceed beyond the Software Strategy Project, NEC in 1980 established the Software Product Engineering Laboratory to lead the company's efforts in software engineering R&D, making this organization part of NEC's central research laboratories. The Software Factory Design Project, started in 1986 under the auspices of the laboratory, developed guidelines for designing actual software factories, from higher-level concepts, such as tool and methodology standardization, to smaller details, such as how to arrange programmers' work spaces or recreation areas.

NEC's Software Quality Control (SWQC) Program dates back to 1978, when a handful of NEC managers established a software quality-control study group. Several specific conclusions came out of their reviews. First, research in software development indicated a strong correlation between quality and productivity, reflecting the manpower needed to fix defects. Hence, they concluded that any revolution in software productivity would require correspondingly dramatic improvements in quality-control practices. Surveys of NEC projects also supported the observation that "human factors," i.e. differences in programmer skills and experience, seemed to be the most

important elements influencing individual performance, and that NEC had to address training more seriously if it were to make major advances in productivity or quality (Mizuno 1983).

NEC management then set up a quality-control program that focused on motivation, teamwork methodologies, training, and other factors affecting individual performance.[21] Since evidence from manufacturing departments indicated that bringing employees together in small groups helped solve quality problems, NEC imported the concept of quality circles. Next, in 1981, NEC created a formal, company-wide organization covering all aspects of software production, management, services, sales, and training, under the SWQC (Software Quality Control) Program. NEC management also found the SWQC practices brought major improvements in productivity through reductions in debugging time, defects, and specification changes (Mizuno 1983).

The Software Problem Strategy Project, another three-year effort launched in 1982, attempted to encourage more standardization in development and quality-control practices, explore various productivity-improvement measures, and establish or formally designate a series of software factories to serve NEC's different product divisions. Under this project, NEC executives decided to act in three areas. First, they carried out a "software production mapping" that consisted of constructing a logistical and organizational layout of programming operations within NEC by product (basic software, industrial systems, business applications, transmission systems, switching software, and microcomputer software), to determine which software houses divisions were using to assist in development and whether divisions needed more help, such as new subsidiaries that might serve as additional software factories. Second, they formalized and systematized procedures for managing software subcontractors. And third, they launched another effort to improve and link software

productivity and quality-assurance measures by establishing a Software Productivity Committee to study documentation control, quality control, software productivity and quality measurements, cost estimation, personnel education, project management, support tools, and production environments.[22]

The centralized laboratory for software-engineering process R&D did not work quite as well as NEC managers had hoped. Some laboratory researchers had become too "academic" in orientation while engineers and SWQC teams in the product divisions seemed to be doing more useful applied studies. To encourage more practical research that better met the needs of divisions, but without eliminating all basic research, a 1987 reorganization moved the basic researchers to NEC's Central Research Laboratories. This involved no organizational change, since the Software Product Engineering Laboratory had been a part of the central labs. However, physically removing the more academic researchers left a group more concerned with applications of new methods and tools. Management then expanded the number of applied researchers and divided them into four areas under the umbrella of a newly created C&C [Computers and Communications] Software Development Group.[23]

The Software Planning Office took charge of running the company-wide software quality-control effort. The Software Engineering Development Laboratory conducted research on tools and integrated development environments, as well as software-engineering management, and established a consulting department to help transfer technology or assist operating divisions and subsidiaries. The C&C Common Software Development Laboratory developed basic-software packages for microcomputers, while the C&C Systems Interface Laboratory worked on compatibility and network technology.

Fujitsu

Fujitsu established a basic software division within its hardware factory in the mid-1970s that closely resembled Hitachi's Software Works in practices and organization except that Fujitsu kept basic hardware development on the same site as basic software development. An important characteristic of Fujitsu's development approach and organization was the gradual integration of controls for product, process, and quality. Direction of Fujitsu's efforts in these areas, as at NEC, came from the Quality Assurance Department in the Numazu Works's Software Division.

According to a chronology the department prepared, these efforts fell into three main phases: prior to 1970, when Fujitsu had no set procedures and managers allowed programmers to test software at their own discretion; 1970-1978, when Fujitsu set up its first product and process standards and formal systems for inspection and quality control; and the period after 1978, when Fujitsu began placing more emphasis on structured design and programming techniques and established the procedures that formed the basis of its current practices. Distinguishing the last phase was a broadening of the Quality Assurance Department's concerns to include not simply testing and documentation conformance, or product evaluation, but analysis of the development process itself. Like Hitachi, Toshiba, and NEC, these practices brought major improvements in quality as well as productivity, with, for example, the number of defects in all outstanding basic-software code supported by Fujitsu dropping to 0.19 in 1977 and then to 0.01 in 1985.[24]

In applications, Fujitsu's decision to create a software factory stemmed from the same need Hitachi, Toshiba, and NEC encountered: to produce a variety of nominally different programs more efficiently, primarily sold with the company's own hardware and basic software. Management began cautiously.

First, it experimented with a centralized organization by setting up a Software Conversion Factory Department in 1977, with approximately 100 personnel. This modified programs customers wanted to run on new machines, which were not compatible with Fujitsu's previous architectures, as well as software originally written for other companies' machines for operation on Fujitsu hardware. Managers believed conversion work was fairly straightforward and that centralization of personnel and equipment would foster standardization and thus dissemination of good methods and tools, making tasks easier to manage and resulting in higher productivity and quality. This seemed feasible especially since, in coordination with the factory establishment, a team of Fujitsu engineers defined a set of structured design and programming techniques as well as detailed procedures for project management, called SDEM (Software Development Engineering Methodology), and introduced support tools for programming in Fortran, called SDSS (Software Development Support System), which Fujitsu quickly replaced with tools for COBOL programming (Murakami et al. 1981).

The conversion factory worked well enough to expand the facility to include program construction by adding another 200 personnel and charging them with turning requirements specifications received from systems engineers into code. Prior to this, Fujitsu had managed systems engineering and program construction in integrated projects, with no separation of these two functions. The process for new development did not work smoothly for all projects. Much like SDC had experienced a few years earlier (but without publicizing this), Fujitsu managers found that many projects depended on close interactions with customers and knowledge of very different requirements, or that writing the application program required access to proprietary information which customers, for security reasons, preferred not to give Fujitsu personnel unless they worked

at the customers' own sites. On other occasions, Fujitsu needed to provide programming services at locations around Japan, again departing from the centralized factory model. In addition, as Fujitsu improved the tools and reuse databases available in the factory, less-experienced programmers became better able to build complete systems on their own, making separation of work and use of the more skilled systems engineers unnecessary.

Rather than abandoning the objective of streamlining software development through a factory approach, Fujitsu improved its system gradually, recognizing that different projects had different optimal processes. The major change consisted of expanding the scope of work in the factory departments to let factory personnel do detailed design and eventually systems design -- for projects where it was difficult or unnecessary to separate these tasks, either for logistical reasons or because factory engineers had the expertise to design and build systems on their own (Figure 3).

Fujitsu introduced other changes. One encouraged systems engineers outside the factory, who initially did surveys and project planning, systems design, and a program's structural design, to leverage their expertise more widely not only by letting the factory personnel do more work but by writing software packages to cover the needs of many users -- with a single design effort. Any packages or pieces of them that Fujitsu could deploy for custom jobs, as is or modified, reduced the need to write new software. In addition, to spread the burden of programming more widely, Fujitsu management established or engaged more subsidiaries and subcontractors, as well as leased methods, tools, and training services to customers of Fujitsu hardware, beginning with SDEM in 1980 (Table 11). Fujitsu also continued to refine the factory's methods and tools as the technology and customer needs evolved.

As of 1988, the Systems Engineering Group consisted of three main areas

with several divisions, departments, and subsidiaries, as well as a research institute (Table 12). One area, the Common Technology Divisions, included the SE [Systems-Engineering] Technical Support Center, the Applications Software Planning Division, and the Office Computer Systems Support Division. The SE Technical Support Center housed the Software Factory Department and a portion of its 1500 programmers, as well as other departments for Systems Development Engineering (technology planning and transfer), Information Support (product information for customers), Systems Engineering Support Services (tools and methods), and the Japanese SIGMA project (a joint company and government effort started in 1985 by Japan's Ministry of International Trade and Industry (MITI) in an attempt to disseminate, through a national communications network and standardization around UNIX as a programming environment, the same type of work stations, support tools, and reusable-software techniques that factories such as Toshiba relied on). The factory built about 20% of new applications done in the Systems Engineering Group as well as handled about 75% of all program conversion work (modifying software to run on new Fujitsu machines). Most of the remaining jobs, approximately 800 small projects per year in the late 1980s, went to approximately three dozen subsidiaries as well as subcontractors outside the factory.

A second area consisted of departments with systems engineers specializing in particular industry applications (finance, insurance, securities, manufacturing, distribution, NTT, scientific, technical, government), so that they had adequate knowledge to specify customer requirements and accurate system designs. The third area, functionally specialized departments of systems engineers, designed management information systems, "value-added networks" for different on-line services, personal-computer systems, and software for new telecommunication firms (the new common carriers or NCCs).

CONCLUSION: EVOLUTION TOWARD FACTORY PRACTICE

This review of major software and computer producers in the U.S. and Japan demonstrates that, whether or not companies adopted the factory label, they clearly moved beyond craft practices and toward more systematic engineering and even factory processes or organizations, recycling reusable components as well as introducing standardized methods and tools, deskilling operations, and thus managing software development systematically across a series of similar projects. The transition to these practices required years of effort and passage through overlapping phases comparable to what firms in other industries encountered as they grew and formalized operations (Robbins 1987: 16-19). In software, however, the first step demanded an unusual conviction on the part of key engineers, division managers, and top executives that software was not an unmanageable technology. This led to the creation of formal organizations and control systems rather than continuing to treat software as a loosely organized service provided free to customers primarily to facilitate hardware sales.

Imposing greater structure on the development process also demanded continual initiatives to introduce and refine elements common in other manufacturing and engineering environments but not in software, at least not in the 1960s and early 1970s: a product focus narrower than simply "programming," to limit the range of problems managers and programmers faced; standards, at least temporary ones and tailored to specific product families, that introduced solutions to recurring problems in the form of methods, procedures, and tools, as well as provided guidelines on expected worker performance to aid budgeting and scheduling; training of employees and managers, to standardize skills though without specifying all tasks for all situations; and development of mechanized or partially automated tools for design, testing, product control,

reuse support, personnel and project management, and other functions, as well as continual refinements of these tools and methods as well as products (Table 13).

To a large degree, factory programs signaled commitments to long-term, integrated efforts -- above the level of individual projects -- to standardize, structure, and support software development along the lines suggested by software-engineering literature since the early 1970s. Some firms have needed to do this more than others, for example, if they were short of experienced personnel and competed in markets where other firms might offer comparable or better software more quickly and cheaply. In Japan during the 1970s, there was a special urgency to improve levels of productivity, quality, and process control to offset shortages of skilled programmers and rapid rises in demand, particularly for complex customized applications programs and lengthy basic software for their new hardware systems, introduced to compete with the IBM System/370.

Every firm attempting to introduce factory concepts and practices encountered problems. IBM found it difficult both to coordinate and standardize programming centers dispersed around the world as well as to manage large facilities with thousands of software personnel centralized in one location. SDC disbanded its software factories after systems engineers had difficulty specifying requirements for one unfamiliar application and project managers in general disliked separating system design from programming. Hitachi in the early 1970s failed to reuse components or introduce one standard methodology for all types of software. NEC relied too heavily for tool and methodology development on a centralized laboratory that became divorced from the needs of operating divisions. Fujitsu gradually modified its applications factory organization to allow factory personnel to do more design work. Even

Toshiba, which located process R&D for software next to its factory and focused on a relatively narrow range of products, had to tolerate variations in practices and needs among different departments. But though each Japanese firm had somewhat different products, emphases, and experiences, Hitachi, Toshiba, NEC, and Fujitsu pursued factory models long after SDC abandoned the effort. And compared to IBM, they appeared to place more emphasis on creating specific facilities with thousands of programmers, many without formal software or computer-engineering backgrounds, and relying on a high level of integration among tools, techniques, training, product designs, and other elements.

Overall, IBM and the major Japanese producers followed more or less similar paths in managing software development. Their efforts began with decisions to create centralized organizations and management-control systems for specific product families, standardize around methods and tools tailored to these products, and then provide partially automated support for development and project management, before proceeding to various refinements as well as more automated and "flexible" tools, capable of handling different languages or tasks. What constituted a factory was difficult to measure precisely, although factories clearly seemed different from job-shop or project-centered approaches in their degree of focus on relatively routine designs, standardized methods and tools, standardized training, and controls and integration of these elements across a series of similar projects.

ACKNOWLEDGEMENTS

This article is based on research elaborated on in Michael A. Cusumano, **Japan's Software Factories**, New York and London, Oxford University Press, 1990. I would like to thank the companies cited in the text who cooperated in this study and especially those managers who participated in interviews. David Finnell, a former master's student at Sloan, provided research assistance on System Development Corporation. Many other colleagues and students at M.I.T. also contributed significantly to the ideas presented here.

Table 1: NATO REPORT ON SOFTWARE-ENGINEERING PROBLEMS (1968)

- Lack of understanding in system requirements on the part of customers and designers.
- Large gaps between estimates of costs and time with actual expenditures due to poor estimating techniques, failure to allow time for changes in requirements, and division of programming tasks into blocks before the divisions of the system are well-enough understood to do this properly.
- Large variations, as much as 26:1 in one study, in programmers' productivity levels.
- Difficulty of dividing labor between design and production (coding), since design-type decisions must still be made in coding.
- Difficulty in monitoring progress in a software project, since "program construction is not always a simple progression in which each act of assembly represents a distinct forward step."
- Rapid growth in the size of software systems.
- Poor communication among groups working on the same project, exacerbated by too much uncoordinated or unnecessary information, and a lack of automation to handle necessary information.
- Large expense of developing on-line production control tools.
- Difficulty of measuring key aspects of programmer and system performance.
- A tradition among software developers of not writing systems "for practical use," but trying to write new and better systems, so that they are always combining research, development, and production in a single project, which then makes it difficult to predict and manage.
- Rapid growth in the need for programmers and insufficient numbers of adequately trained and skilled programmers.
- Difficulty of achieving sufficient reliability (reduced errors and error tolerance) in large software systems.
- Dependence of software on hardware, which makes standardization of software difficult across different machines.
- Lack of inventories of reusable software components to aid in the building of new programs.
- Software maintenance costs often exceeding the cost of the original system development.

Source: Naur and Randell.

Table 2: IBM SYSTEM JOURNAL TOPICAL ANALYSIS, 1962-1986

Note: Each year and topic refers to one article. The list excludes articles describing individual products and refers to generic tools and methods.

SOFTWARE-ENGINEERING TOOLS

Design

1967 modeling programs
1981 software simulation as a tool for usable product design
1982 DB/DC data dictionary for businesssystems planning
1983 abstract design and program translator

Coding

1968 Fortran subroutine package
1975 program generator
1979 automatic programming for energy management
1985 tools for building advanced user interfaces

Testing/Product Evaluation

1962 general purpose systems simulator
1963 generation of input data
1963 laboratory data-taking system
1964 systems simulation
1965 expanded general purpose simulator
1969 simulating operating systems
1970 automatic generation of test cases
1971 Fortran extended-precision library
1972 management business simulation in APL
1975 performance measurement tools for VM/370
1984 automatic generation of random self-checking tests
1984 design and use of a program execution analyzer

Phase/Project Management

1985 automating the software development process

Documentation

1985 project automated librarian

SOFTWARE-ENGINEERING METHODS:

Design

1963 construction of minimal project networks
1963 requirements generation
1963 programming notation in systems design
1969 three-value design verification system
1972 techniques for developing analytic models
1974 structured design
1974 elements of probability for system design
1976 top-down development using a program design language
1976 HIPO and integrated program design
1982 strategies for information requirements determination

1982 technique for assessing external design of software
1983 simple architecture for consistent application program design
1983 software reliability analysis
1984 architecture prototyping in the software engineering environment
1985 a process-integrated approach to defect prevention
1985 strategies for problem prevention

Coding

1965 tables, flow charts, and program logic
1968 system for implementing interactive applications
1969 data management
1969 coding for error control
1971 formal description of programming languages

Testing/Product Evaluation

1965 testing real-time system programs
1965 reliability of polymorphic systems
1969 synthetic job for measuring system performance
1971 time-sharing performance criteria and measurement
1972 scientific computing service evaluation
1972 a perspective on system evaluation
1973 user program performance in virtual storage system
1973 experimental evaluation of system performance
1974 performance analysis of the Skylab terminal system
1975 performance analysis of virtual memory time-sharing systems
1975 testing in a complex systems environment
1976 design and code inspections to reduce errors
1982 cost-benefit analysis and the data-managed system
1983 testing of interactive applications
1983 performance/availability measurement for IBM information network
1985 quality emphasis at IBM's Software Engineering Institute

---

Maintenance

1985 requirements methodology for software system enhancements

---

Phase/Project Management

1963 project evaluation and selection
1972 chief programmer team
1973 forecasting techniques
1975 productivity of computer-dependant workers
1976 model of large program development
1977 method of programming measurement and estimation
1978 measuring programming quality and productivity
1979 managing VM/CMS systems for user effectiveness
1980 management of software engineering
1980 systems management
1982 use of data flow to improve application development productivity
1985 programming productivity measurement system for system/370
1985 worldwide systems engineering
1985 the system planning grid
1985 programming process architecture
1985 programming process study
1986 software engineering

Programming Environment

1978 IBM's Santa Teresa Laboratory
1981 Human Factors Center at San Jose
1982 towards an integrated development environment

Source: This is based on a study of indices, tables of contents, and actual articles in the **IBM Systems Journal**. My thanks to Wilson Wong, an undergraduate in Electrical Engineering and Computer Science at M.I.T., for his assistance in reviewing the journal articles for a graduation thesis project under my supervision.

Table 3: IBM SYSTEM JOURNAL ARTICLES SUMMARY, 1962-1986

	TOOLS			METHODS			TOTAL		
	1960s	1970s	1980s	1960s	1970s	1980s	1960s	1970s	1980s
Design	1		3	4	5	7	5	5	10
Coding	1	2	1	4		1	5	3	1
Testing	6	4	2	3	9	4	9	13	6
Process Management			1	1	7	9	1	7	10
Documen-tation			1				0	0	1
Mainte-nance						1	0	0	1
Programming Environment					1	2	0	1	2

Source: Table 2.5.

Table 4: MAJOR JAPANESE SOFTWARE FACTORIES

Key: BS = Operating Systems, Database Management Systems, Language Utilities, and Related Basic Software
App = General Business Applications
RT = Industrial Real-Time Control Applications
Tel = Telecommunications Software (Switching, Transmission)

Notes: All facilities develop software for mainframes or minicomputers. Employee figures refer to 1988 or 1989 estimates, based on company interviews and site visits.

Est.	Company	Facility/Organization	1988-1989 Products	1988-1989 Employees
1969	Hitachi	Hitachi Software Works	BS	5000
1976	NEC	Software Strategy Project		
		Fuchu Works	BS	2500
		Mita Works	RT	2500
		Mita Works	App	1500
		Abiko Works	Tel	1500
		Tamagawa Works	Tel	1500
1977	Toshiba	Fuchu Software Factory	RT	2300
1979	Fujitsu	Systems Engineering Group	App	4000
		(Kamata Software Factory		1500)
1983	Fujitsu	Numazu Software Division (Numazu Works est. 1974)	BS	3000
1985	Hitachi	Systems Design Works	App	6000
		(Systems Engineers		4000)
		(Programming Personnel		2000)

Source: Company data, site visits, and manager interviews.

Table 5: SOFTWARE PROCESS CONTROL IN HITACHI, 1967-1976

1967	Completion of a system for computing labor and machine hours for software development (Kanagawa Works).
1969	Establishment of standard times for software development (Software Works).
1971	Establishment of programmer ability coefficients and amendments of standard times.
	Completion of an estimation and budget system using standard times and a simulation system for resource planning.
	Completion of a PERT Network System (PNET), an automatic diagramming system for schedule control.
	Implementation of a manual system for time-series analysis of test processing and quality control.
1972	Completion of a system for budget-vs.-actual expenditure control for work-hours and machine-hours for each project and department.
	Implementation of a manual system for document schedule control.
1973	Implementation of a manual system for job process control.
	Implementation of a manual system for productivity analysis and control.
1974	Completion of a productivity analysis system for each project and department.
1975	Implementation of a manual system for defect factor analysis and quality control.
1976	Development of a time-series statistical program for forecasting defects (FRCST).
	Establishment of a standard scheduling system.
	Introduction of structured design methods.

Source: Hitachi Ltd., Software Works, "Table 1.2 Background and Conditions Affecting CAPS Development," Undated memorandum.

Table 6: ELEMENTS OF THE TOSHIBA SOFTWARE FACTORY

Combined Tool, Methodology, and Management Systems
-- Project progress management system
-- Cost management system
-- Productivity management system
-- Quality assurance system with standardized quality metrics
-- A standardized, baseline management system for design review, inspection and configuration management
-- Software tools, user interfaces and tool maintenance facilities
-- Existing software library and maintenance support for this
-- Technical data library
-- Standardized technical methodologies and disciplines
-- Documentation support system
Personnel Systems
-- Quality circle activities
-- Education programs
-- Career development system
Physical Infrastructure
-- Specially designed work spaces

Source: Matsumoto 1987, p. 155.

Table 7: MAJOR DIVISIONS AND PRODUCTS AT TOSHIBA FUCHU WORKS

Information Processing and Control Systems Division:
Information Systems for Public Facilities
Industrial Automation Systems
Industrial-Use Computers
Information-Processing and Control Systems Equipment
Energy Systems Division:
Digital Control Systems for Power Generating Stations
Automated Power Supply and Distribution Systems
Computer-Aided Business Operations Systems for Electric Power Utilities
Control Equipment for Power Plants
Electronics Equipment for Power Transmission Systems
Protection Equipment for Electric Power Systems
Advanced Technology-Applied Power Systems
Industrial Equipment Division:
Switch gear
Factory Automation Components
Power Electronics Equipment
Mechatronics Equipment
Elevators/Escalators
Rolling Stock Systems
Transportation/Carrying Systems
Electronic Modules
Printed Wiring Boards Division
Printed Wiring Boards
High-Density Hybrid Functional Circuits

Source: Toshiba Corporation, "Toshiba Fuchu Works," 1987, p. 5.

Table 8: TOSHIBA SOFTWARE WORKBENCH EVOLUTION

Initial Development	Support Tool	Improvement/Automation Focus
1976-1977	SWB-I	Programming Environment (programming, assembly, compiling, debugging in the source level, project files, program generation, program reusing and link edit)
1978-1980	SWB-II	Test Environment
1980-1985	SWB-P	Project Management
1980-1985	SWB-Q	Quality Assurance
1981-1983	SWB-III	Design Support (requirements specification, software design description, and documentation)
1981-1984	SWB-IV	Program Maintenance
1986-1988	Various	Document Preparation Expert System [for Design] Reusable Software Integrated Database

Source: Matsumoto 1987: 12; "Development of SWB for Toshiba Fuchu Software Factory," transparency copy received from Y. Matsumoto, 9/4/87; Matsumoto interview, 1988.

Table 9: PRODUCTIVITY & REUSE AT TOSHIBA SOFTWARE FACTORY

Notes: EASL = Debugged and Delivered Equivalent Assembler Source Lines Per Programmer Per Month, Averaging All Projects in the Factory and Including All Phases and Manpower (Requirements Analysis through Maintenance)

Index = Based on 1972 EASL productivity estimate (1230)

Change = Percent increase or decrease over previous year

Reuse % = Percent of delivered lines of code taken from existing software with little or no modifications (usually no more than 20% of a given module)

New Code = EASL x [(100 - Reuse %)/100]

Quality = Defects Per 1000 Lines of Delivered Code Converted to EASL

Year	Total EASL Delivered Per Person Per Month	Index/Change (100) (%)		Reuse %	New Code (EASL)	Defects Per 1000 EASL	Employees (All Phases)
PRE-FACTORY ESTIMATES:							
1972	1230	100	--	Data Not Available			
1973	1390	113	+13				
1974	1370	111	- 2				
1975	1210	98	-12				
1976	1390	113	+15				
POST-FACTORY ESTIMATES:							
1977	Data Not Available						
1978	1684	137	--	--	--	7 to 20	1200
1979	1988	162	+18	13	1730		1500
1980	2072	168	+ 4	16	1740		1700
1981	2443	199	+18	29	1735		2050
1982	2595	210	+ 6	26	1920		2100
1983	2763	225	+ 7	41	1630		2150
1984	2931	238	+ 6	45	1612		2250
1985	3130	254	+ 7	48	1612	0.2 to 0.05	2300

Source: The pre-factory figures are from K. H. Kim, "A Look at Japan's Development of Software Engineering Technology," **Computer**, May 1983, p. 33. The 1978-1985 figures are estimated from index data presented in graph form in an internal Toshiba memo, "The Factory History Data Based on the Values of 1978," received by M. Cusumano from Dr. Yoshihiro Matsumoto, September 1988, using actual 1985 values published in Matsumoto 1987, p. 5.

Table 10: NEC SOFTWARE FACTORY IMPLEMENTATION

YEAR	INITIATIVE	FOCUS/OUTCOMES
1974	Basic Software Development Division	Organizational separation of software from hardware development
1976-1979	Software Strategy Project	Standardization of data collection, tool and structured-programming methodology for basic and applications software throughout the NEC group, with the objectives of raising productivity and quality
1980	Software Product Engineering Laboratory	Centralization of process and tool R&D for dissemination to divisions and subsidiaries
1981	Software Quality Control (SWQC)	Establishment of a group-wide methodology, training program, and control measures for improving software quality, including quality circle activities
1982-1985	Software Problem Strategy Project	1) "Mapping" of software development activities 2) Subcontracting management 3) Software productivity improvement
1986	Software-Factory Design Project	Establishment of Hokuriku Software Development Center, based on ergonomic principles and other software-factory concepts
1987	C&C Software Development Group	Reorganization of the Software Product Engineering Laboratory and expansion of applied research

Source: Fujino and Azuma interviews, 7/28/86; Mizuno interview 8/25/88.

Table 11: APPLICATIONS SOFTWARE DEVELOPMENT AT FUJITSU

1964	Establishment of Systems Department.
1970	Centralization of applications systems development at Information Processing System Laboratory in Kamata, Tokyo.
1976	Establishment of Software Engineering Team within Systems Department to develop SDEM (Software Development Engineering Methodology).
1977	Establishment of Software Conversion Factory and SDEM standards. EPG (Executive Planning Guide) and C-NAP (Customer Needs Analysis Procedure) commercialized for in-house use and sale.
1978	Introduction of SDSS ((Software Development Support System).
1979	Establishment of Software Factory Department in Kamata.
1980	SDEM standards commercialized for sale.
1981	YAC (Yet Another Control chart) developed for design notation. Commercialization of PARADIGM (methodology and tool for developing reusable programs).
1982	Commercialization of BAGLES (Banking Application Generator's Library for Extensive Support). Commercialization of ADAM (Applications Development and Management System)
1983	SIMPLE tool-set introduced for commercial sales, beginning with LINDA (Logical Information Support Tool of Dataset) and DBSP (Database Support Program). YACII and YPS (YAC II Programming System) developed. SDT (SDEM Design Technique) developed.
1984	On-line SDEM developed.
1985	ACS-APG (ACS Application Program Generator) from the SIMPLE series commercialized.
1987	DRAGON (Driver and Stub Generator for On-line and Batch Test) from the SIMPLE series commercialized. SDAS (Systems Development Architecture and Support Facilities) commercialized. EPGII/CNAPII commercialized, adding to 1977 version design concept and data-analysis capabilities

Source: Fujitsu internal data.

Table 12: FUJITSU SYSTEMS ENGINEERING GROUP ORGANIZATION

COMMON TECHNOLOGY DEPARTMENTS

SE Technical Support Center
- -- Systems Development Engineering Department
- -- **Software Factory Department**
- -- Information Support Center Department
- -- Systems Engineering Support Services Department
- -- SIGMA Project Systems Promotion Office

Office Computer Systems Support Service Division

Application Software Planning Division
- -- Application Software Planning Department
- -- Software Distribution Department

INDUSTRY-RELATED DEPARTMENTS

Finance
Insurance/Securities
Manufacturing/Distribution
Scientific/Technical
NTT
Government/Mass-Communication

FUNCTIONAL DEPARTMENTS

Management Information Systems
VAN (Value-Added Network) Systems Engineering
Information Network Systems Engineering
Personal Computer Systems Engineering
NCC (New Common Carriers) Systems Engineering

SYSTEMS ENGINEERING SUBSIDIARIES

Industry-Related
Functional
Regional

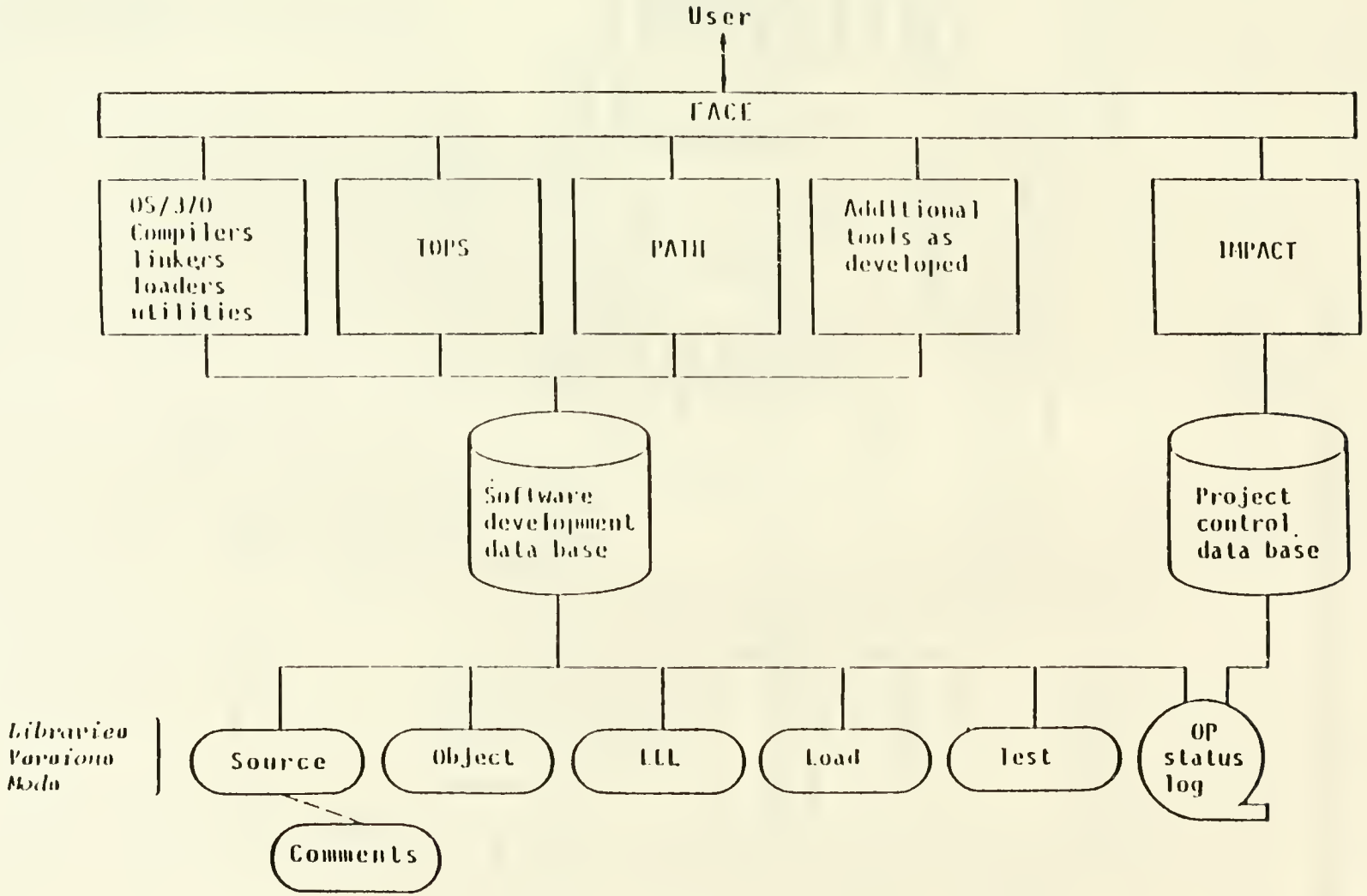
FUJITSU RESEARCH INSTITUTE FOR ADVANCED INFORMATION SYSTEMS AND ECONOMICS

Source: Nikkei Computer, 13 October 1986, p. 79, updated to 1989.

Table 13: PHASES OF FACTORY STRUCTURING IN SOFTWARE

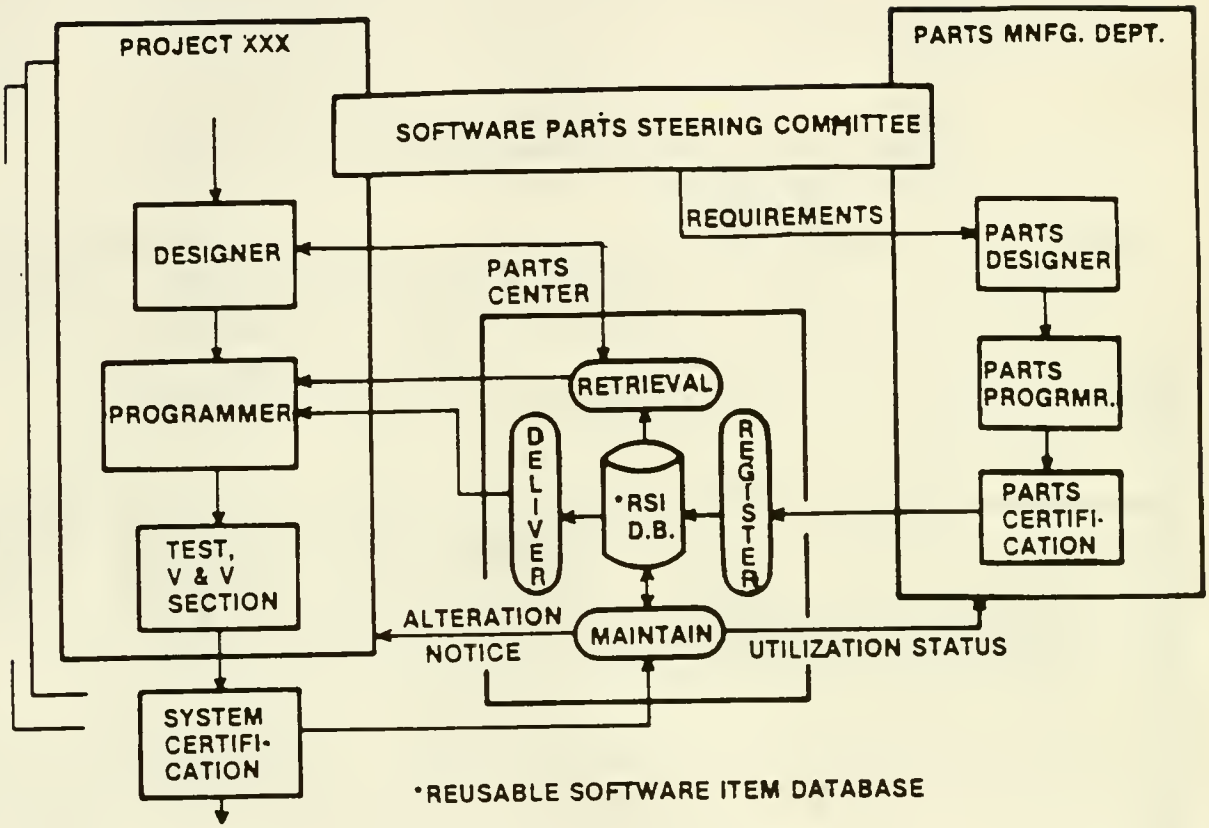
Phase I: (Mid-1960s to Early 1970s)	**Formalized Organization and Management Structure** Factory Objectives Established Product Focus Determined Process Data Collection and Analysis Begun Initial Control Systems Introduced
Phase II: (Early 1970s to Early 1980s)	**Technology Tailoring and Standardization** Control Systems and Objectives Expanded Standard Methods Adopted for Design, Coding, Testing, Documentation, Maintenance On-Line Development Through Terminals Program Libraries Introduced Integrated Methodology and Tool Development Begun Employee Training Programs to Standardize Skills
Phase III: (Late 1970s)	**Process Mechanization and Support** Introduction of Tools Supporting Project Control Introduction of Tools to Generate Code, Test Cases, and Documentation Integration of Tools with On-line Databases and Engineering Work Benches Begun
Phase IV: (Early 1980s)	**Process Refinement and Extension** Revisions of Standards Introduction of New Methods and Tools Establishment of Quality Control and Quality Circle Programs Transfer of Methods and Tools to Subsidiaries, Subcontractors, Hardware Customers
Phase V: (Mid-1980s)	**Integrated and Flexible Automation** Increase in Capabilities of Existing Tools Introduction of Reuse-Support Tools Introduction of Design-Automation Tools Introduction of Requirements Analysis Tools Further Integration of Tools Through Engineering Work Benches
Phase VI: (Late 1980s)	**Incremental Product/Variety Improvement** Process & Reliability Control, Followed By: Better Functionality & Ease of Use More Types of Products

Figure 1: SDC SOFTWARE FACTORY ARCHITECTURE



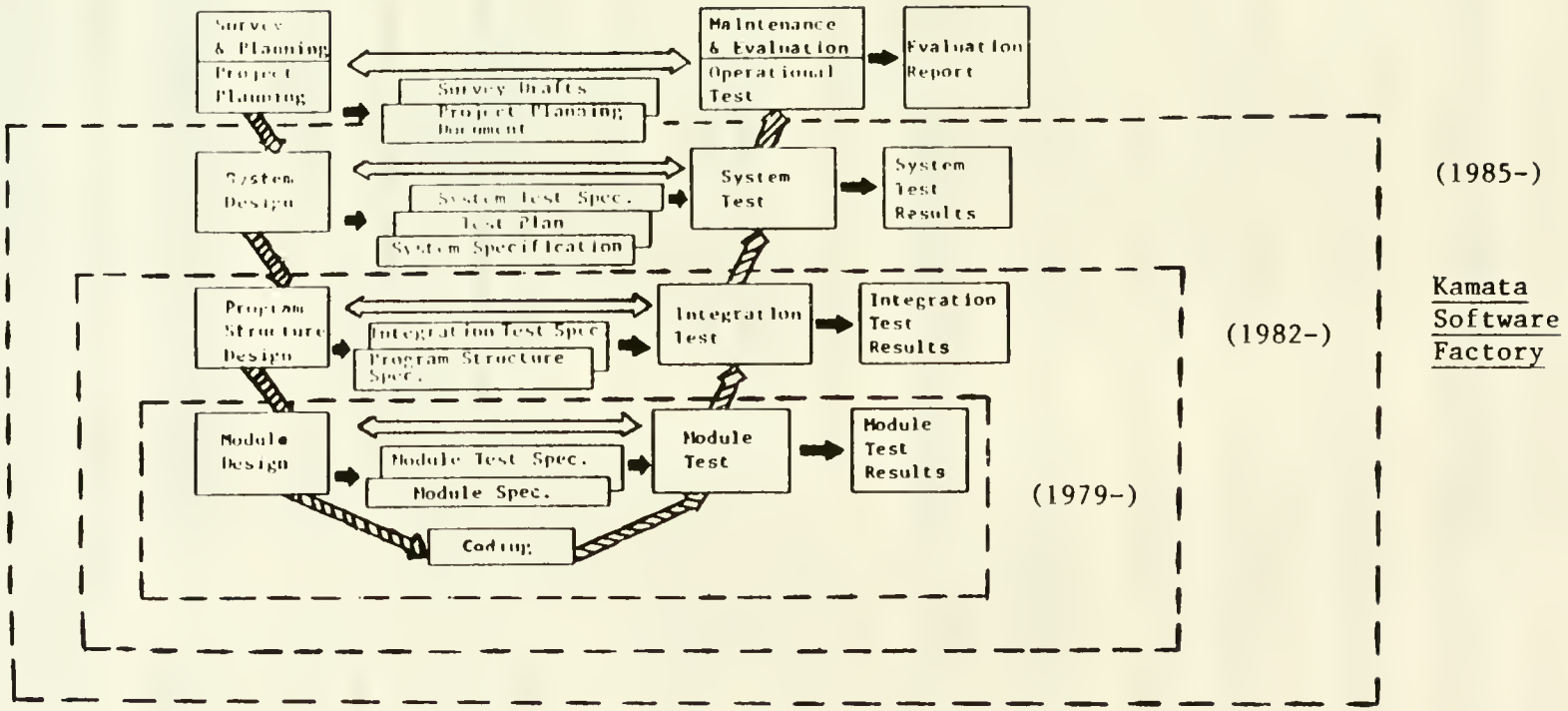
Source: Bratman and Court, 1977, p. 129

Figure 2: TOSHIBA REUSE PROMOTION SYSTEM



Source: Matsumoto 1987, p. 173

Figure 3: FUJITSU APPLICATIONS FACTORY PROCESS FLOWS



Source: Narafu and Marakami Interviews.

REFERENCES

Abdel-Hamid, Tarek 1984. "The Dynamics of Software Development Project Management: An Integrative Systems Dynamic Perspective," unpublished Ph.D. dissertation, M.I.T. Sloan School of Management.

Abernathy, William J. 1978. **The Productivity Dilemma: Roadblock to Innovation in the Automobile Industry**, Baltimore, Johns Hopkins University Press.

____ and James M. Utterback 1978. "Patters of Technological Innovation," **Technology Review**, 80, June/July, pp. 40-47.

____ 1982. "Patterns of Industrial Innovation," in Michael L. Tushman and William L. Moore, **Readings in the Management of Innovation**, New York, Pitman, pp. 97-108.

Arden, Bruce W. (ed.) 1980. **What Can Be Automated?**, Cambridge, MA, MIT Press.

Baum, Claude 1981. **The System Builders: The Story of SDC**, Santa Monica, Cal., System Development Corporation.

Bemer, R.W. 1969. "Position Papers for Panel Discussion -- The Economics of Program Production," **Information Processing 68**, Amsterdam, North-Holland.

Boehm, Barry W. 1976. "Software Engineering," **IEEE Transactions on Computers**, Vol. C-25, No. 12 (December), 1226-1241.

Borum, Finn 1987. "Beyond Taylorism: The IT-Specialists and the Deskilling Hypothesis," Copenhagen School of Economics, Computer History (CHIPS) Working Paper.

Bratman, Harvey, and Terry Court 1975. "The Software Factory," **Computer** (May), 28-37.

____. "Elements of the Software Factory: Standards, Procedures, and Tools," in Infotech International Ltd., **Software Engineering Techniques**, Infotech International Ltd., Berkshire, England, pp. 117-143.

Brooks, Jr., Frederick P. 1975. **The Mythical Man-Month: Essays on Software Engineering**, Reading, MA, Addison-Wesley.

_____ 1987, "No Silver Bullet: Essence and Accidents of Software Engineering," **IEEE Computer**, April, 10-19.

Chandler, Jr., Alfred D. 1977. **The Visible Hand: The Managerial Revolution in American Business**, Cambridge, M.A., Harvard University Press.

Conte, S.D. et al. 1986. **Software Engineering Metrics and Models**, Menlo Park, CA., Benjaman/Cummings.

DeLamarter, Richard Thomas 1986. **Big Blue: IBM's Use and Abuse of Power**, New York, Dodd, Mead.

Fisher, Franklin M., James W. McKie, and Richard B. Mancke 1983. **IBM and the U.S. Data Processing Industry: An Economic History**, New York, Praeger.

Frank, Werner L. 1983. **Critical Issues in Software**, New York, John Wiley and Sons.

Freeman, Peter ed. 1987. **Software Reusability**, Washington, D.C., IEEE Computer Society.

Fujino, Kiichi 1984. "Software Development for Computers and Communications at NEC," **Computer** (November), 57-62.

Hagi, Yoichi et al. 1986. "Shisutemu kaihatsu shien sofutouea 'EAGLE'" (Integrated Software Development and Maintenance System 'EAGLE'), **Hitachi hyoron**, 68, 5, 29-34.

Hitachi Ltd. 1979, **Softutouea Kojo 10 nen no ayumi** (A 10-year history of the Software Works), Yokohama, Hitachi Ltd.

Horowitz, Ellis, and John B. Munson 1984. "An Expansive View of Reusable Software," **IEEE Transactions on Software Engineering**, SE-10, 5, 477-487.

Hounschell, David A. 1984. **From the American System to Mass Production, 1800-1932**, Baltimore, Johns Hopkins University Press.

Hunke, Horst, ed. 1981. **Software Engineering Environments**, Amsterdam, North-Holland.

Jones, Capers 1986. **Programming Productivity**, New York, McGraw-Hill.

Kobayashi, M. et al. 1983. "ICAS: An integrated Computer Aided Software Engineering System," **IEEE Digest of Papers--Spring '83 COMPCON**, 238-244.

Kobayashi, Masakazu and Yoshihiko Aoyama, "Sofutouea seisan gijutsu no saikin no koko" (Current Topics in Software Engineering), **Hitachi hyoron**, 68, 5, 1-6.

Matsumoto, Yoshihiro 1981, "SWB System: A Software Factory," in H. Hunke, ed., **Software Engineering Environments**, North-Holland, Amsterdam, pp. 305-318.

____ 1984. "Management of Industrial Software Production," **Computer** (February), 59-71.

____ 1987, "A Software Factory: An Overall Approach to Software Production," in Peter Freeman, ed., **Software Reusability**, Washington, D.C., IEEE Computer Society, pp. 155-178.

Matsumura, Kazuo et al. 1987, "Trend Toward Reusable Module Component" Design and Coding Technique 50SM," Tokyo, **Proceedings of the Eleventh Annual International Computer Software and Applications Conference--COMPSAC**, IEEE Computer Society Press, October 7-9.

McCue, G.H. 1978. "IBM's Santa Teresa Laboratory: Architectural Design for Program Development," **IBM Systems Journal**, 17, 1,

McIlroy, M.D. 1969. "Mass Produced Software Components," in Peter Naur and Brian Randell, eds., **Software Engineering: Report on a Conference Sponsored by the NATO Science Committee**, Scientific Affairs Division, NATO, Brussels, January 1969, pp. 151-155.

Miyazoe, Hidehiko et al. 1980. "Apurikeshon shisutemu no koritsu-teki sekkei giho 'HIPACE'" (Software Engineering Methodology for Development of Application Systems 'HIPACE'), **Hitachi hyoron**, 62, 12, 15-20.

Mizuno, Yukio 1983. "Software Quality Improvement," **Computer** (March), 66-72.

Murakami, Noritoshi et al. 1981. "SDEM and SDSS: Overall Approach to Improvement of the Software Development Process," in H. Hunke, ed., **Software Engineering Environments**, Amsterdam, North-Holland, pp. 281-293.

Nakabayashi, Sen et al. 1986. "C&C Satellite Office and Networks," **NEC Research and Development**, 81 (April), 8-18.

Naur, Peter and Brian Randell, eds. 1969. **Software Engineering: Report on a Conference Sponsored by the NATO Science Committee**, Brussels, Scientific Affairs Division, NATO.

Ramamoorthy, C.V., et al. 1984. "Software Engineering: Problems and Perspectives," **IEEE Computer** (October), 191-209.

Robbins, Stephen P. 1987. **Organization Theory: Structure, Design, and Applications**, Englewood Cliffs, N.J., Prentice-Hall.

Schware, Robert 1989. **The World Software Industry and Software Engineering**, Washington, D.C., The World Bank, Technical Paper No. 104.

Shimoda , Hirotsugu 1986. **Sofutouea kojo** (Software factories), Tokyo, Toyo Keizai Shimposha.

Shooman, Martin 1983. **Software Engineering: Design, Reliability, and Management**, New York, McGraw-Hill.

Thayer, R.H. 1979. "Modeling a Software Engineering Project Management System," unpublished Ph.D. dissertation, University of California at Santa Barbara.

Toshiba Corporation 1987, "Toshiba Fuchu Works," Tokyo.

Tsuchiya, Nobuyoshi et al. 1986. "A Control and Management System for the C&C Satellite Office," **NEC Research and Development**, 81 (April), 19-23.

U.S. Department of Commerce 1984. **A Competitive Assessment of the U.S. Software Industry**, Washington, D.C., International Trade Administration.

Woodward, Joan 1965. **Industrial Organization: Theory and Practice**, London, Oxford University Press.

Yoshida, Tadashi 1985. "Sofutouea no keiryo-ka" (Quantifying software), **Joho shori**, 26, 1, 48-51.

NOTES

1. Interview with James Frame, 10/13/88, formerly manager of IBM programming centers at Endicott, N.Y., Raleigh, N.C., and at Santa Teresa Laboratory.

2. Frame interview.

3. Frame interview.

4. Interviews with Frederick George, Manager, Network Systems Design, IBM Corporation, Raleigh, N.C., 1/6/87; and Mark Harris, Manager, Intermediate Processor Development, IBM Corporation, Endicott, N.Y., 12/12/86 and 12/16/86.

5. Letter from Glenn Bacon, former director of programming at the Santa Teresa Laboratory, to Cusumano, 4/25/88.

6. Interview with John B. Munson, Vice President and General Manager, Unisys Houston Operations, 10/4/87 and 10/5/1987. Also, Baum 1981: 224.

7. Interviews with Munson and Terry Court, Manager, Resources Development Laboratory, GM-Hughes Aircraft, Inc., 6/22/89.

8. Court interview.

9. Interviews with Munson and Court. Also, Baum 1981: 222,

10. Interview with Ronald Atchley, Director, Software Engineering Group, Unisys/System Development Corporation, 3/27/87 and 4/23/87.

11. Performance data is from Hitachi 1979: 114, updated to 1985 by Kanji Shibata, Manager, Engineering Department, Hitachi Software Works, 7/23/86.

12. Shibata interview. See also Hitachi 1979: 5.

13. Hitachi 1979: 114-115; and interview with Shibata.

14. Interview with Kazuyuki Sakata, formerly Deputy General Manager of the Hitachi Software Works and subsequently Managing Director, Nippon Business Consultant, Ltd., 9/10/85.

15. Toshiba Corporation, "Trend of Application Software Size," transparency copy, received 15 March 1988.

16. Interviews with Yoshihiro Matsumoto, former Fellow Scientist, Toshiba Corporation, 9/4/87 and 8/24/88, and currently Professor of Information Systems, Kyoto University.

17. This assumes a 1:3 conversion for the EASL data, a fair rate for the data given that 60% of Toshiba code was in Fortran; about 20% was in assembler or similar intermediate level languages, requiring a lower conversion rate; and another 20% appeared in other languages (Matsumoto 1987).

18. Productivity numbers can be found in numerous sources, including Jones 1986: 92-114; Conte et al. 1986: 251, 270; Shooman 1983: 438-445; Brooks 1975: 88-94; U.S. Department of Commerce 1984: 11.

19. Interview with Yoshio Ikeda, Program Manager, Power-Generation Control Computer Systems Department, Fuchu Works, Toshiba, 7/17/89.

20. Interview with Yozo Hirai, Manager of the Quality Assurance Department in the Basic Software Division, NEC Corporation, 9/8/87; also, Tsuchiya et al. 1986 and Nakabayashi et al. 1986.

21. This next discussion is based on interviews with Kiichi Fujino, Vice President, NEC Corporation, 7/28/86 and 9/8/87, and Motoei Azuma, former manager of the Software Engineering Department in the Software Product Engineering Laboratory, NEC Corporation, 10/1/86, current a Professor of Engineering, Waseda University.

22. Interviews with Azuma and Fujino, 7/28/86.

23. Interview with Yukio Mizuno, Senior Vice President, NEC Corporation, 8/25/88.

24. This data is from Yoshida 1985: 49, updated to 1985 by Tadashi Yoshida, Manager, Quality Assurance Department, Software Division, Fujitsu, in a personal interview, 9/7/87.

4867 054

Date Due 3-1-90

JUN 1 1990

APR 08 1991

JUN 26 1992

Lib-26-67

3 9080 00579053 7